第 2 章

文件

引言

感谢：Mark Lutz，*Programming Python* 和 *Python Quick Reference* 的作者，*Learning Python* 的合著者之一

任何一个有经验的程序员接触一门新语言时，都会首先在该语言的工具箱中寻找文件的相关工具。因为处理外部文件是一种非常实用和常见的任务，该语言的文件处理接口设计的好坏，在很大程度上决定着这门语言工具的实用性。

正如本章各节将要介绍那样，Python 在这方面非常优秀。Python 中对文件的支持，体现在很多层次上：从内建的 open 函数（标准文件对象类型的一个同义词），到标准库模块提供的一些特定的工具，比如 os，再到遍布网络的各种第三方提供的实用工具。一言以蔽之，Python 强大的文件工具库提供了各种方法来访问文件。

文件基础

在 Python 中，文件对象是内建类型 file 的实例。内建函数 open 会创建并返回一个文件对象。第一个参数是一个字符串，指定了文件路径（文件名之前有一个可选的目录路径）。open 的第二个参数也是个字符串，指定了打开文件的模式。比如：

```
input = open('data', 'r')
output = open('/tmp/spam', 'w')
```

open 接受由斜线字符（/）分隔开的目录和文件名构成的文件路径，而完全不管操作系统本身的倾向。在不使用斜线的系统中，可以使用反斜线字符（\），不过如果没有什么好理由的话最好不要这么做。反斜线在字符串中的文本表示不太美观，你得用两个反斜线来表示单个斜线字符或者用“原”字符串（raw string）。如果文件路径参数中没有包括文件的路径，则该文件被认为处于当前工作路径中（这个路径可能游离于 Python 模块的搜索路径之外）。

至于模式参数，使用'r'意味着以文本模式读取文件；这是默认的值而且常常被忽略，所以 open 也可以只使用第一个参数。其他常见的模式是'rb'，表示以二进制模式读取文件，'w'表示创建文件并以文本模式写文件，'wb'表示创建并以二进制模式写文件。'r'的一种变体是'rU'，意味着将以支持“通用换行符”（universal newline）的文本模式读取文件：'rU'模式可以用一种独立于该文件所用的断行约定的方式来读取文件，可以是 UNIX 方式，也可以是 Windows 方式，甚至是（老的）Mac 方式。（今天的 Mac OS X 从各方面来讲都可以算是 UNIX，但前几年的 Mac OS 9 还完全不一样）

对于那些非类 UNIX 平台（non-UNIX-like platforms），文本模式和二进制模式之间的区别非常重要，这和那些系统所用的行结束标记有关。当你以二进制模式打开一个文件，Python 知道它无须关心行结束标记；它只是在文件和内存中的字符串之间移动字节，不需要任何翻译转化。但当你在一个非类 UNIX 系统中以文本模式打开一个文件时，Python 知道它必须在字符串所用的“\n”换行符和当前系统所用的行结束标记之间做翻译和转化工作。只要你能正确地指定你打开的文件的模式，你的所有的 Python 代码总可以依赖“\n”作为行结束标记。

一旦有了一个文件对象，你就可以执行该对象的各种 I/O 操作，我们接下来会介绍那些操作。当你完成了处理，应该对该对象调用 close 方法来完成收尾工作，关闭所有和这个文件对象的联系：

```
input.close( )
```

在一些短的脚本中，用户常常忽略这个步骤，因为当一个文件对象在垃圾回收阶段被收回的时候（在主流的 Python 中这意味着该文件会被立即关闭，但其他的一些重要的 Python 实现，比如 Jython 和 IronPython，则有其他更宽松的垃圾回收策略），Python 会自动关闭该文件对象。不管怎样，用完文件对象之后尽可能马上关闭文件是一个好的编程习惯，尤其是在写一些大型程序的时候，不然你总会让一些无用的打开的文件对象占着内存资源。注意，Try/finally 在确保文件被关闭时非常好用，即使函数抛出了一个无法处理的异常，并因此而终止。

在写入文件的时候，使用 write 方法：

```
output.write(s)
```

假如 s 是一个字符串，当 output 是用文本模式打开的时候，可以将 s 看成是字符串，当 output 以二进制模式打开的时候，则可以将 s 看做是字节串。文件还有其他的一些与写入有关的方法，比如 flush，可以发送出缓存中所有的数据，还有 writelines，一次调用可以将整个字符串序列写入。不过，write 仍然是最常用的方法。

从文件中读取数据比把数据写入文件更常见，也涉及更多的内容，因此文件对象的有关读取的方法要比写入的方法多。readline 方法可从一个文本文件中读取并返回一行文本数据。考虑这样一个循环：

```
while True:
    line = input.readline( )
    if not line: break
    process(line)
```

在 Python 中，这曾经是从一个文件读取并处理每行数据的最惯用的方式，但现在已经不是了。另一个过时的方式是使用 readlines 方法，一次读完整个文件，并返回一个各行数据的列表：

```
for line in input.readlines():
     process(line)
```

readlines 方法只有在物理内存足够用的情况下才会很有用。如果文件非常庞大，readlines 可能会失败，或者性能急剧降低（虚拟内存不足，操作系统会将物理内存中的数据复制到磁盘上）。在现在的 Python 中，只需对这个文件对象执行一个循环，每次取得一行并处理，这样可获得更好的性能和效率：

```
for line in input:
     process(line)
```

当然，也不总是逐行的读取一个文件。可以读取文件中的一些或者所有的字节，特别是在用二进制模式读取的情况下。对于二进制数据，行是一个没有意义的概念。根据这种情况，可以使用 read 方法。当不指定参数时，read 会读取并返回文件中所有剩余的字节。当 read 被传入一个整数参数 N 并调用时，它读取并返回下 N 个字节（或者所有剩余的字节，如果剩余的字节数少于 N）。其他的值得一提的是 seek 和 tell 方法，支持对文件的随机访问。在处理包含很多固定长度记录的二进制文件时，这些方法很常用。

可移植性和灵活性

从表面看，Python 对文件的支持非常直接。然而，在你仔细阅读本章的代码之前，我想指出关于 Python 文件支持的两个重要方面：代码的可移植性和接口的灵活性。

请记住，Python 中的大多数文件接口都是完全跨平台的。这种设计的重要性怎么赞美也不为过。比如，在一个在目录树中搜索所有文件中的文本的脚本，可以一点代码不改地运行在各个平台上：只需要将脚本文件复制到目标机器上。我一直都是这么做的——完全不用关心操作系统的差异性。由于 Python 的强大的可移植性，运行平台几乎成为一个无足轻重的问题。

还有，Python 的文件处理接口往往并不局限在只用于真实的物理文件，这也让我很吃惊。实际上，大多数文件工具也可以用于那些暴露的接口与文件对象类似的任何类型的对象。因此，文件读取者只需要关心读取方法，文件写入者则只需关心写入方法。只要目标对象实现了期望的协议，一切工作都可以平滑自然地进行。

比如，假设你要写一个通用的文件处理函数，像下面这样，传入一个函数来处理输入

文件中的每行数据：

```
def scanner(fileobject, linehandler):
     for line in fileobject:
          linehandler(line)
```

如果将这个函数写入一个模块文件，并将该文件放入包括在 Python 搜索路径（sys.path）中的一个“目录”里，就可以在任何时候调用该函数来逐行扫描文本文件，无论何时。为了展示这一点，下面给出一个客户脚本，只是简单地打印出每行的第一个单词：

```
from myutils import scanner
def firstword(line):
     print line.split( )[0]
file = open('data')
scanner(file, firstword)
```

至此为止，一切都很好；我们完成了一个小的可复用的软件组件。但请注意，在 scanner 函数中并没有对类型有任何假设，只要是能被逐行迭代的满足接口要求的对象就可以。比如，假如你以后想测试一些来自于字符串对象的输入，而不是真实的物理文件。标准的 StringIO 模块，以及等价但更快速的 cStringIO，提供了一种适用的封装和类似的接口：

```
from cStringIO import StringIO
from myutils import scanner
def firstword(line): print line.split( )[0]
string = StringIO('one\ntwo xxx\nthree\n')
scanner(string, firstword)
```

StringIO 对象完全兼容文件对象，即插即用，所以，scanner 从内存中的一个字符串对象中读取三行文本，而不是从一个真正的外部文件中读取。完全不用更改原先的 scanner 来完成任务，只需传递一个正确的对象。为了更具通用性，还可以实现一个类并提供 scanner 期望的接口：

```
class MyStream(object):
      def __iter__(self):
          # 获取并返回字符串
          return iter(['a\n', 'b c d\n'])
from myutils import scanner
def firstword(line):
      print line.split( )[0]
object = MyStream( )
scanner(object, firstword)
```

这次，当 scanner 尝试读取文件时，它实际调用的是在类中实现的_ _iter_ _方法。在实践中，这个方法可以借助 Python 标准工具从各种源抓取文本：一个交互的用户，一个弹出式的图形界面输入框，一个 shelve 对象，一个 SQL 数据库，一页 XML 或者 HTML，一个网络 socket 等。关键点在于，scanner 根本就不知道也不关心究竟是什么类型的对象实现了它期望的接口，以及这些接口究竟在背地里干什么。

面向对象语言的程序员都知道一个重要的概念，多态。被处理对象的类型决定了究竟

进行什么样的操作——比如 scanner 中的循环迭代。而在 Python 中，对象的接口，而非类型，才是真正的连接器。这种机制造成的实际效果是一个函数的应用面往往要比你设想的广得多。特别是如果你有一些静态类型语言如 C 或 C++的背景，对此会有更深的体会。就好像我们在 Python 中突然获得了免费的 C++模板。Python 的强大的动态类型带来的一个副产品是，代码具有了与生俱来的灵活性。

当然，代码的可移植性和灵活性在 Python 的开发中无处不在，不仅仅局限于文件接口。这两者都是语言本身的特性，这段文件处理脚本只是简单地继承了这些特性。其他的 Python 的优点，还包括代码容易编写和易读，当需要修改你的文件处理程序的时候，你就能体会到这些好处了。Python 除了这些值得赞美的优点之外，在本章和本书中还有很多精彩的内容和细节值得你去发掘和探索。阅读愉快！

2.1 读取文件

感谢：Luther Blissett

任务

你想从文件中读取文本或数据。

解决方案

最方便的方法是一次性读取文件中的所有内容并放置到一个大字符串中：

```
all_the_text = open('thefile.txt').read( )     # 文本文件中的所有文本
all_the_data = open('abinfile','rb').read( )   # 二进制文件中的所有数据
```

为了安全起见，最好还是给打开的文件对象指定一个名字，这样在完成操作之后可以迅速关闭文件，防止一些无用的文件对象占用内存。举个例子，对文本文件读取：

```
file_object = open('thefile.txt')
try:
    all_the_text = file_object.read( )
finally:
    file_object.close( )
```

不一定要在这里用 Try/finally 语句，但是用了效果更好，因为它可以保证文件对象被关闭，即使在读取中发生了严重错误。

最简单、最快，也最具 Python 风格的方法是逐行读取文本文件内容，并将读取的数据放置到一个字符串列表中：

```
list_of_all_the_lines = file_object.readlines( )
```

这样读出的每行文本末尾都带有“\n”符号；如果你不想这样，还有另一个替代的办法，比如：

```
list_of_all_the_lines = file_object.read( ).splitlines( )
list_of_all_the_lines = file_object.read( ).split('\n')
list_of_all_the_lines = [L.rstrip('\n') for L in file_object]
```

最简单最快的逐行处理文本文件的方法是，用一个简单的 for 循环语句：

```
for line in file_object:
    process line
```

这种方法同样会在每行末尾留下“\n”符号；可以在 for 循环的主体部分加一句：

```
line = line.rstrip('\n')
```

或者，你想去除每行的末尾的空白符（不只是'\n'\），常见的办法是：

```
line = line.rstrip( )
```

讨论

除非要读取的文件非常巨大，不然一次性读出所有内容放进内存并进一步处理是最快和最方便的办法。内建函数 open 创建了一个 Python 的文件对象（另外，也可以通过调用内建类型 file 创建文件对象）。你对该对象调用 read 方法将读出所有内容（无论是文本还是二进制数据），并放入一个大字符串中。如果内容是文本，可以选择用 split 方法或者更专用的 splitlines 将其切分成一个行列表。由于切分字符串到单行是很常见的需求，还可以直接对文件对象调用 readlines，进行更方便更快速的处理。

可以直接对文件对象应用循环语句，或者将它传递给一个需要可迭代对象的处理者，比如 list 或者 max。当它被当做一个可迭代对象处理时，一个被打开并被读取的文件对象中的每一个文本行都变成了迭代子项（因此，这也只适用于文本文件）。这种逐行迭代的处理方式很节省内存资源，速度也不错。

在 UNIX 或者类 UNIX 系统中，比如 Linux，Mac OS X，或者其他 BSD 变种，文本文件和二进制文件其实并没有什么区别。在 Windows 和老的 Macintosh 系统中，换行符不是标准的'\n'，而分别是'\r\n'和'\r'。Python 会帮助你把这些换行符转化成'\n'。这意味着当你打开二进制文件时，需要明确告诉 Python，这样它就不会做任何转化。为了达到这个目的，必须传递'rb'给 open 的第二个参数。在类 UNLX 平台上，这么做也不会有什么坏处，而且总是区分文本文件和二进制文件是一个好习惯，当然在那些平台上这并不是强制性的要求。不过这些好习惯会让你的程序具有更好的可读性，也更易于理解，同时还能具有更好的平台兼容性。

如果不确定某文本文件会用什么样的换行符，可以将 open 的第二个参数设定为'rU'，指定通用换行符转化。这让你可以自由地在 Windows、UNIX（包括 Mac OS X），以及其他的老 Macintosh 平台上交换文件，完全不用担心任何问题：无论你的代码在什么平台上运行，各种换行符都被映射成'\n'。

可以对 open 函数产生的文件对象直接调用 read 方法，如解决方案中给出的第一个代码片段所示。当你这么做的时候，你在完成读取的同时，也失去了对那个文件对象的引用。在实践中，Python 注意到了这种当场即时失去引用的情况，它会迅速关闭该文件。然而，更好的办法仍然是给 open 产生的结果指定一个名字，这样当你完成了处理，可以显式地自行关闭该文件。这能够确保该文件处于被打开状态的时间尽量的短，即使是在 Jython，IronPython 或其他变种 Python 平台上（这些平台的高级垃圾回收机制可能会推迟自动回收，不像现在的基于 C 的 Python 平台，CPython 会立刻执行回收）。为了确保文件对象即使在处理过程发生错误的情况下仍能够正确关闭，应该使用 try/finally 语句，这是一种稳健而严谨的处理方式。

```
file_object = open('thefile.txt')
try:
     for line in file_object:
          process line
finally:
     file_object.close( )
```

注意，不要把对 open 的调用放入到 try/finally 语句的 try 子句中（这是初学者很常见的错误）。如果在打开文件的时候就发生了错误，那就没有什么东西需要关闭，而且，也没有什么实质性的东西绑定到了 file_object 这个名字上，当然也就不应该调用 file_object.close()。

如果选择一次读取文件的一小部分，而不是全部，方式就有点不同了。下面给出一个例子，一次读取一个二进制文件的 100 个字节，一直读到文件末尾：

```
file_object = open('abinfile', 'rb')
try:
     while True:
          chunk = file_object.read(100)
          if not chunk:
               break
          do_something_with(chunk)
finally:
     file_object.close( )
```

给 read 方法传入一个参数 N，确保了 read 方法只读取下 N 个字节（或更少，如果读取位置已经很接近文件末尾的话）。当抵达文件末尾时，read 返回空字符串。复杂的循环最好被封装成可复用的生成器（generator）。对于这个例子，我们只能将其逻辑的一部分进行封装，这是因为生成器（generator）的 yield 关键字不被允许出现在 try/finally 语句的 try 子句中。如果要抛弃 try/finally 语句对文件关闭的保护，我们可以这么做：

```
def read_file_by_chunks(filename, chunksize=100):
     file_object = open(filename, 'rb')
     while True:
```

```
            chunk = file_object.read(chunksize)
            if not chunk:
                 break
            yield chunk
      file_object.close( )
```

一旦 read_file_by_chunks 生成器完成，以固定长度读取和处理二进制文件的代码就可以写得极其简单：

```
for chunk in read_file_by_chunks('abinfile'):
     do_something_with(chunk)
```

逐行读取文本文件的任务更为常见。只需对文件对象应用循环语句，如下：

```
for line in open('thefile.txt', 'rU'):
     do_something_with(line)
```

为了 100%确保完成操作之后没有无用的已打开的文件对象存在，可以将上述代码修改得更加严密稳固：

```
file_object = open('thefile.txt', 'rU'):
try:
     for line in file_object:
          do_something_with(line)
finally:
     file_object.close( )
```

更多资料

第 2.2 节；*Library Reference* 和 *Python in a Nutshell* 中关于内建 open 函数和文件对象的内容。

2.2 写入文件

感谢：Luther Blissett

任务

你想写入文本或者二进制数据到文件中。

解决方案

下面是最方便的将一个长字符串写入文件的办法：

```
open('thefile.txt', 'w').write(all_the_text)  # 写入文本到文本文件
open('abinfile', 'wb').write(all_the_data)    # 写入数据到二进制文件
```

不过，最好还是给文件对象指定个名字，这样你就可以在完成操作之后调用 close 关闭

文件对象。比如，对一个文本文件：

```
file_object = open('thefile.txt', 'w')
file_object.write(all_the_text)
file_object.close( )
```

可是，很多时候想写入的数据不是在一个大字符串中，而是在一个字符串列表（或其他序列）中。为此，应该使用 writelines 方法（不要望文生义，这个方法并不局限于行写入，而且二进制文件和文本文件都适用）：

```
file_object.writelines(list_of_text_strings)
open('abinfile', 'wb').writelines(list_of_data_strings)
```

当然也可以先把子串拼接成大字符串（比如用''.join）再调用 write 写入，或者在循环中写入，但直接调用 writelines 要比上面两种方式快得多。

讨论

要创建一个用于写入的文件，必须将 open（或文件对象）的第二个参数指定为“w”以允许写入文本数据，或者“wb”以允许写入二进制数据。当你试图写入而不是读取文件的时候，你更应该重视 2.1 节提到的关闭文件的建议。只有在文件被正确关闭之后，你才能确信数据被写入了磁盘，而不是暂存于内存中的临时缓存中。

分批次将数据写入文件的应用，甚至比分批次从文件中读取数据的应用更为常见。只需准备妥当字符串或者字符串序列，然后反复地调用 write 或 writelines 即可。每个 write 操作都会在文件末尾增添新数据，紧随已经写入的旧数据。当你完成了写入工作，可调用 close 方法来关闭文件对象。如果你能够一次准备好所有需要写入的数据，可以调用 writelines 来一次性完成写入任务，这样更快也更简单。但如果你每次只能准备好一部分数据，那么最好分批次写入。比起另一个方法，即，先在内存中建立一个大的临时数据序列用于存储所有的数据，然后用 writelines 一次性将所有数据写入文件，分批次写入明显要更好一些。读取和写入这两种操作，在一次性大数据处理和分批次小数据处理这两个方面，性能和方便程度都表现出很大的区别。

当用“w”（或“wb”）选项打开一个文件准备写入数据的时候，文件中原有的数据都将被清除；即使在打开之后迅速关闭，得到的仍然是一个空文件。如果你想把新数据添加在原有的数据之后，应该用“a”（或“ab”）选项来打开文件。还有一些更高级的选项可允许你对同一个文件同时进行读取和写入操作，见第 2.8 节中提到的“r+b”选项，那实际上也是高级选项中最常用的一个。

更多资料

第 2.1 节；第 2.8 节；*Library Reference* 和 *Python in a Nutshell* 中关于内建 open 函数和文件对象的内容。

2.3 搜索和替换文件中的文本

感谢：Jeff Bauer, Adam Krieg

任务

需要将文件中的某个字符串改变成另一个。

解决方案

字符串对象的 replace 方法提供了字符串替换的最简单的办法。下面的代码支持从一个特定的文件（或标准输入）读取数据，然后写入一个指定的文件（或标准输出）：

```
#!/usr/bin/env python
import os, sys
nargs = len(sys.argv)
if not 3 <= nargs <= 5:
     print "usage: %s search_text replace_text [infile [outfile]]" % \
          os.path.basename(sys.argv[0])
else:
     stext = sys.argv[1]
     rtext = sys.argv[2]
     input_file = sys.stdin
     output_file = sys.stdout
     if nargs > 3:
          input_file = open(sys.argv[3])
     if nargs > 4:
          output_file = open(sys.argv[4], 'w')
     for s in input_file:
          output_file.write(s.replace(stext, rtext))
     output.close( )
     input.close( )
```

讨论

本节给出的解决方案非常简单，但那也正是精彩的地方——如果简单的东西已经够用了，为什么要用复杂的东西？正如开始的“shebang”（shell 头描述）行所示，这个脚本是一个简单的主脚本，而不是那些被用来导入的模块，它直接运行在一个 shell 命令行提示中。脚本检查传递给它的参数以确定要搜索的文本，用于替代的文本，输入文件（默认是标准输入），输出文件（默认是标准输出）。然后循环遍历输入文件中的每一行，完成了对每行文本的字符串替换之后，再写入到输出文件。这就结束了。准确点说，最后还关闭了所有的文件。

如果内存充裕到能够轻松放入两份输入文件（一份是原版的，另一份是完成了替代之后的，因为字符串是不能被改变的，所以必须有两个拷贝），我们就可以进一步地提高速度。一次性完成对所有内容的处理，而不用循环。今天的低端计算机至少都有 256MB

以上的内存，处理一个 100MB 左右的文件并不是什么问题，而且处理超过 100MB 的文件的情况也很少。所以，我们也可以用一行语句替换掉那个循环：

```
output_file.write(input_file.read( ).replace(stext, rtext))
```

正如你所看到的，简单得不得了。

更多资料

参看 *Library Reference* 和 *Python in a Nutshell* 中关于内建 open 函数，文件对象，字符串的 replace 方法。

2.4 从文件中读取指定的行

感谢：Luther Blissett

任务

你想根据给出的行号，从文本文件中读取一行数据。

解决方案

Python 标准库 linecache 模块非常适合这个任务：

```
import linecache
theline = linecache.getline(thefilepath, desired_line_number)
```

讨论

对这个任务而言，标准的 linecache 模块是 Python 能够提供的最佳解决工具。当你想要对文件中的某些行进行多次读取时，linecache 特别有用，因为 linecache 会缓存一些信息以避免重复一些工作。当你不需要从缓存中获得行数据时，可以调用模块的 clearcache 函数来释放被用作缓存的内存。当磁盘上的文件发生了变化时，还可以调用 checkcache，以确保缓存中存储的是最新的信息。

linecache 读取并缓存你指定名字的文件中的所有文本，所以，如果文件非常大，而你只需要其中一行，为此使用 linecache 则显得不是那么必要。如果这部分可能是你的程序的瓶颈，可以使用显式的循环，并将其封装在一个函数中，这样可以获得速度上的一些提升，像这样：

```
def getline(thefilepath, desired_line_number):
    if desired_line_number < 1: return ''
    for current_line_number, line in enumerate(open(thefilepath, 'rU')):
        if current_line_number == desired_line_number-1: return line
    return ''
```

唯一需要注意的细节是 enumerate 从 0 开始计数，因此，既然我们假设 desired_line_number 参数从 1 开始计算，需要在用==比较的时候减去 1。

更多资料

参见 *Library Reference* 和 *Python in a Nutshell* 中的 linecache 模块；*Perl Cookbook* 8.8。

2.5 计算文件的行数

感谢：Luther Blissett

任务

需要计算一个文件中有多少行。

解决方案

对于尺寸不大的文件，最简单的方式是将文件读取放入一个行列表中，然后计算列表的长度即可。假设文件路径由变量 thefilepath 指定，那么以这种方式实现的代码如下：

```
count = len(open(thefilepath, 'rU').readlines( ))
```

对于非常大的文件，这种简单的处理方式极有可能会很慢，甚至会失败。如果你确实担心大文件的问题，用循环来计数是一个可行的办法：

```
count = -1
for count, line in enumerate(open(thefilepath, 'rU')):
    pass
count += 1
```

如果行结束标记是“\n”（或者含有“\n”，就像 Windows 平台），我们还有一个更巧妙的，对于大文件也更快的方式：

```
count = 0
thefile = open(thefilepath, 'rb')
while True:
    buffer = thefile.read(8192*1024)
    if not buffer:
        break
    count += buffer.count('\n')
thefile.close( )
```

给 open 的‘rb’参数是必要的，如果你追求速度，那么没有那个参数，这段代码在 Windows 上的运行可能会比较慢。

讨论

如果有外部程序提供文件行统计的功能，比如类 UNIX 平台中的 wc –l，你当然也可以

选择使用它们（比如，通过 os.popen）。但是，如果能够实现自己的行计算程序，代码通常会更简单、更快，也更具移植性。对于那些大小比较适合的文件，一次全部读取到内存中再处理是最简单的方式。对于这种文件，用 len 对 readlines 返回的结果计算长度即可获取行数。

如果文件大到超过了可用的内存（比如，几百兆字节，对于今天的典型的个人计算机来说），这种最简单的方式将慢得根本无法接受。操作系统费尽九牛二虎之力，试图把文件内容放入虚拟内存，而且这个过程还可能失败，如果交换区空间耗尽，虚拟内存也无以为继。假设在一个典型的个人计算机上，装有 256MB 内存和无限制的虚拟磁盘，你尝试一次性读取超过 1GB 或 2GB 的文件，需要当心这个过程中可能发生的错误，不过这跟你使用的操作系统还有一些关系。（一些操作系统在极端的高负载压力下处理虚拟内存会比其他系统脆弱得多）在这个场合中，用循环来处理文件对象，如本节解决方案所示，会更好一些。内建的 enumerate 函数会自行计算行数，无须你用代码明确指定。

一次读取适量的字节，并计算其中的换行符，这是本节第三个处理方式的思路。这可能不是那么直观，而且也不能很完美地跨平台，但它可能是最快的办法（可以将它和 *Perl Cookbook* 8.2 节比较一下）。

然而，大多数时候，性能不是那么重要。如果性能的确值得考虑，你的第一感直觉也往往不能告诉你程序中真正耗时的代码段是哪个部分，事实上，你绝不应该相信直觉——而应该进行基准测试。比如，有个典型的中等大小的 UNIX syslog 文件，18MB 略多一点，230 000 行文本：

```
[situ@tioni nuc]$ wc nuc
231581 2312730 18508908 nuc
```

考虑下面的基准测试框架脚本，bench.py：

```
import time
def timeo(fun, n=10):
     start = time.clock( )
     for i in xrange(n): fun( )
     stend = time.clock( )
     thetime = stend-start
     return fun.__name__, thetime
import os
def linecount_w( ):
     return int(os.popen('wc -l nuc').read( ).split( )[0])
def linecount_1( ):
     return len(open('nuc').readlines( ))
def linecount_2( ):
     count = -1
     for count, line in enumerate(open('nuc')): pass
     return count+1
def linecount_3( ):
```

```
        count = 0
        thefile = open('nuc', 'rb')
        while True:
            buffer = thefile.read(65536)
            if not buffer: break
            count += buffer.count('\n')
        return count
for f in linecount_w, linecount_1, linecount_2, linecount_3:
        print f.__name__, f( )
for f in linecount_1, linecount_2, linecount_3:
        print "%s: %.2f"%timeo(f)
```

首先，我将各种方法统计行数的结果打印出来，以确保没有什么错误或反常发生（众所周知，行统计任务会因为一点小错而失败）。然后，通过控制和计时函数 timeo，我再将各个任务都运行 10 次，并观察结果。在一台可靠的老机器上，我的程序得出如下结果：

```
[situ@tioni nuc]$ python -O bench.py
linecount_w 231581
linecount_1 231581
linecount_2 231581
linecount_3 231581
linecount_1:4.84
linecount_2:4.54
linecount_3:5.02
```

正如你所见的，性能差异几乎可以忽略：用户对这类辅助性任务从来都感觉不出 10% 的性能差异。然而，最快的方式却是简单朴实地循环遍历每一行（我的测试环境是，一台老旧但可靠的个人计算机，运行着一个流行的 Linux 发行版本），最慢的竟然是更具技巧性的逐次读取数据并计算换行符的方式。在实践中，除非需要处理非常大的文件，我一般总会选择最简单的方式（本节提供的第一个方法）。

准确地衡量代码的性能（要比盲目使用一些复杂的方式并寄希望能提高性能好的多）非常重要——重要到 Python 标准库要专门提供一个模块，timeit，用来测量程序的速度。我建议你用 timeit，而不是用自己创造的一些测量方法，就像我在这里所做的。但是这个测试方法我多年前就在使用了，甚至比 timeit 模块出现在 Python 标准库的时间还早，所以，在这个例子中我没有用 timeit 也算情有可原吧。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于文件对象，内建 enumerate 函数，os.popen 以及 time 和 timeit 模块；*Perl Cookbook* 8.2。

2.6 处理文件中的每个词

感谢：Luther Blissett

任务

你想对一个文件中的每个词做一些处理。

解决方案

完成这个任务的最好的办法是使用两重循环，一个用于处理行，另一个则处理每一行中的每个词：

```
for line in open(thefilepath):
     for word in line.split( ):
          dosomethingwith(word)
```

for 语句假定了词是一串非空的字符，并由空白字符隔开（和 UNIX 程序 wc 一样）。如果词的定义有变化，还可以使用正则表达式。比如：

```
import re
re_word = re.compile(r"[\w'-]+")
for line in open(thefilepath):
     for word in re_word.finditer(line):
          dosomethingwith(word.group(0))
```

在此例中，词被定义为数字字母，连字符或单引号构成的序列。

讨论

如果还需要其他的对词的定义，当然也需要使用不同的正则表达式。外层关于文件行的循环，则不用改变。

通常把迭代封装成迭代器对象是个好主意，这种封装也很常见和易于使用，如下：

```
def words_of_file(thefilepath, line_to_words=str.split):
     the_file = open(thefilepath):
     for line in the_file:
          for word in line_to_words(line):
               yield word
     the_file.close( )
for word in words_of_file(thefilepath):
     dosomethingwith(word)
```

这个方式可以清晰有效地将两部分内容分开：一个是怎么迭代所有的元素（本例中，指的是文件中的词），另一个是要对每个元素做什么处理。一旦你将迭代操作的部分封装在一个迭代器对象中（常常表现为一个生成器（generator）），你就可以只使用一个 for 语句来完成迭代操作了。可以在你的程序各处重复地使用这个迭代器，而且如果需要维护代码的话，你只需要修改一处——即迭代器的定义和实现部分，而不用到处寻找需要修改的部分。这种好处，类似于任何其他语言中正确地定义和使用了函数，就不必到处复制粘贴代码段。通过 Python 的迭代器，可充分复用循环控制结构。

通过重构将循环放入一个生成器，我们还获得其他两个小小的增强——文件被显式地确保关闭了，行文本被划分成单词的方式也更通用了（默认使用字符串对象的 split 方法，但是却给了指定其他方式的可能性）。比如，如果我们需要用正则表达式来取词，可以对 words_of_file 再做一层包装：

```
import re
def words_by_re(thefilepath, repattern=r"[\w'-]+"):
     wre = re.compile(repattern)
     def line_to_words(line):
          for mo in wre.finditer(line):
               return mo.group(0)
     return words_of_file(thefilepath, line_to_words)
```

这里，我们也给出了一种默认词定义的正则表达式定义，当然如果有必要使用其他不同的词定义，也可以传入不同的表达式。过度追求通用化是一种有害的诱惑，但是基于经验所做的一些适度的通用化总是能够事半功倍。采用一个函数，接受可选的参数，并在参数为默认值时提供一些合适的值，是一种快速而方便的实现通用化的方法。

更多资料

第 19 章关于迭代器和生成器的更多内容；*Library Reference* 和 *Python in a Nutshell* 中关于文件对象和 re 模块；*Perl Cookbook* 8.3。

2.7 随机输入/输出

感谢：Luther Blissett

任务

给定一个包含很多固定长度记录的大二进制文件，你想读取其中某一条记录，而且还不需要逐条读取记录。

解决方案

一条记录相对于文件头部的偏移字节，就是这条记录的长度再乘以记录的条数（正整数，从 0 开始计数）。因此，可以直接将读取位置设置在正确的点上，然后读取数据。比如，如果每条记录长度是 48 字节长，则从二进制文件中读取第 7 条记录的方法如下：

```
thefile = open('somebinfile', 'rb')
record_size = 48
record_number = 6
thefile.seek(record_size * record_number)
buffer = thefile.read(record_size)
```

注意，第 7 条，也就是 record_number 是 6：记录条数从 0 开始统计。

讨论

本节的方法适用于包含相同长度的记录的文件（通常是二进制的），而不适用于普通的文本文件。为了表明这一点，本节给出的代码在调用 seek 之前，给 open 函数传递了一个“rb”参数以指明读取二进制文件。只要被读取的文件是作为二进制文件被打开的，在最终关闭文件之前，就可以根据需要随意地使用 seek 和 read 方法——不一定要正好在执行 seek 之前打开文件。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于文件对象的章节；*Perl Cookbook* 8.12。

2.8 更新随机存取文件

感谢：Luther Blissett

任务

给定一个包含很多固定长度记录的大二进制文件，你想读取其中某一条记录，并且修改该条记录的某些字段的值，然后写回到文件中。

解决方案

读取记录，解包，执行任何需要的数据更新，然后将所有字段重新组合成记录，接着找到正确的位置，最后再写入。见如下代码：

```
import struct
format_string = '8l'                    # 或者说，一条记录是 8 个 4 字节整数
thefile = open('somebinfile', 'r+b')
record_size = struct.calcsize(format_string)
record_numberb
thefile.seek(record_size * record_number)
buffer = thefile.read(record_size)
fields = list(struct.unpack(format_string, buffer))
#进行计算，并修改相关的字段，然后:
buffer = struct.pack(format_string, *fields)
thefile.seek(record_size * record_number)
thefile.write(buffer)
thefile.close( )
```

讨论

本节的方法适用于包含相同长度的记录的文件（通常是二进制的），而不适用于普通的文本文件。而且，每条记录的长度由一个结构化的格式化字符串来定义，如代码所示。一个典型的格式化字符串，比如，“8l”，指明每条记录是由 8 个 4 字节整数构成，每个

整数都有个被指定的值而且可以被解包到一个 Python int 类型的对象中。在这个例子中，fields 变量被绑定到了一个 8 个整数的列表上。注意，struct.unpack 返回的是一个元组。由于元组是不可改变的，通过计算完成数据更新之后只能重新绑定 fields 变量。而列表则是可改变的，每个字段可以根据需要重新绑定。因此，为了方便，绑定 fields 的时候我们显式地要求一个列表，同时确保不改变这个列表的长度。对这个例子而言，列表中需要恰好含有 8 个整数，否则当我们利用值为“8l”的 format_string 来打包时，struct.pack 就会抛出异常。如果记录的长度不一，本节的方法也不适用。

为了迅速定位记录的起始地址，可以选择使用相对位置寻址方式，而不是用计算出的 record_size*record_number 偏移字节：

```
thefile.seek(-record_size, 1)
```

传递给 seek 方法的第二个参数值（1），告诉文件对象定位到相对当前位置的某处（此处，需要回退一些字节，因为我们给了第一个参数一个负值）。seek 的默认寻址方式是在文件中用绝对偏移量来定位（从文件开头开始计算）。也可以给 seek 的第二个参数传个 0 值，显式地要求使用默认的行为方式。

不用正好在第一次使用 seek 之前打开文件，也不用调用 write 之后就马上关闭文件。一旦正确地打开了文件（作为需要更新的二进制文件，不是文本文件），在关闭文件之前，可以根据需要对此文件进行任意次更新。在此展示这些调用是为了强调打开文件做随机存取更新的关键所在，同时也是为了再次提醒读者，完成所有操作之后迅速关闭文件的重要性。

这文件需要更新（同时允许读和写）。这正是传递给 open 的“r+b”这个参数的意思：打开文件用于读写，但并不隐式地对文件内容做任何转化，因为这是一个二进制文件。（对于 UNIX 和类 UNIX 系统，“b”部分并不是必需的，但是为了清晰，却是值得推荐的。而且，对某些平台来说，指定这个部分是非常关键的，比如 Windows）如果你准备从头开始创建这个文件，而且还要反复定位，读取并更新一些记录，同时还不用关闭此文件并重新打开，可以将 open 的第二个参数指定为“w+b”。然而，这样的奇怪要求我还从来没见过。通常二进制文件是在第一次创建之后（通过“wb”打开，写入数据，然后关闭文件），再用“r+b”重新打开并进行更新的。

虽然本节方法只适用于记录长度一致的文件，不过，适应更高级需求的可能性也存在：一个单独的“索引文件”，提供了另一个“数据文件”中的每条记录的长度和偏移量的信息。这种连续的不定长记录的方式已经不太流行了，但是在过去却是很重要的方式。如今，我们遇到的也就是文本文件（各种类型，不过 XML 越来越多），数据库，偶尔还会碰到记录长度固定的二进制文件。如果你确实需要处理这种带索引的连续的二进制文件，代码也不会改动太多，只需要从索引文件中读取记录长度 record_size 和偏移量，并把两者作为参数传递给 thefile.seek，不需要如本节解决方案所做的那样，自己计算偏移。

更多资料

Library Reference 和 *Python in a Nutshell* 关于文件对象和 struct 模块；*Perl Cookbook* 8.13。

2.9 从 zip 文件中读取数据

感谢：Paul Prescod、Alex Martelli

任务

你想直接检查一个 zip 格式的归档文件中部分或者所有的文件，同时还要避免将这些文件展开到磁盘上。

解决方案

zip 文件是一种流行的跨平台的归档文件。Python 标准库提供了 zipfile 模块来简便地访问这种文件：

```
import zipfile
z = zipfile.ZipFile("zipfile.zip", "r")
for filename in z.namelist( ):
     print 'File:', filename,
     bytes = z.read(filename)
     print 'has', len(bytes), 'bytes'
```

讨论

Python 能直接处理 zip 文件中的数据。可以直接看到归档目录中的所有子项，并且直接处理这些“数据文件”。解决方案中给出的代码片段能够列出归档文件 Zipfile.zip 中所有文件的名字和长度。

zipfile 模块现在还不能处理分卷 zip 文件和带有注释的 zip 文件。注意，要使用 r 作为标志参数，而不是 rb，虽然 rb 看起来挺自然的（特别是在 Windows 下）。对于 zipfile，它的标志与 open 所用的打开一个文件的标志不太一样，它根本不认识 rb。r 标志可以应付所有平台上的各种 zip 文件。如果 zip 文件中包含一些 Python 模块（也即.py 或者.pyc 文件），也许还有一些其他的（数据）文件，可以把这个文件的路径加入到 Python 的 sys.path 中，并用 import 语句来导入处于这个 zip 文件中的模块。下面给出一个玩具示例，这只是一个自包含的、纯粹的展示型的例子，它会凭空创建一个 zip 文件，并从中导入一个模块，最后再删除该文件——仅仅是为了向你展示如何做到这一切的：

```
import zipfile, tempfile, os, sys
handle, filename = tempfile.mkstemp('.zip')
os.close(handle)
z = zipfile.ZipFile(filename, 'w')
```

```
z.writestr('hello.py', 'def f( ): return "hello world from "+__file__\n')
z.close( )
sys.path.insert(0, filename)
import hello
print hello.f( )
os.unlink(filename)
```

运行这个脚本会产生一些类似这样的输出：

```
hello world from /tmp/tmpESVzeY.zip/hello.py
```

除了展示 Python 从 zip 文件中导入模块的能力，这段代码还显示了怎样制造出一个临时文件，以及怎样使用 writestr 方法来向 zip 文件中添加一个成员，你甚至不用事先在磁盘上创建这个成员。

注意，从 zip 文件导入模块所用的路径会被看做是个目录。（在这个特定的例子里，路径是/tmp/tmpESVzeY.zip，不过，由于我们处理的是一个临时文件，因此每次运行脚本这个值都会不同，具体取决于你用的系统）具体地说，在模块 hello 这个被导入的模块中，全局变量__file__的值是/tmp/tmpESVzeY.zip/hello.pya 这样一种伪路径，它由被看做“目录”的 zip 文件路径加上位于 zip 文件中的 hello.py 的相对路径组合而成。如果从 zip 文件中导入模块，处于 zip 文件中的模块的数据文件将通过相对路径获取，需要适应从 zip 文件导入模块的这种特性，因为你无法用 open 函数来打开一个“伪路径”以获取文件对象：要想读取或者写入 zip 文件中的文件，必须使用 Python 标准库的 zipfile 模块，如同解决方案给出的代码那样。

更多的关于从 zip 文件中导入模块的资料，请参看第 16.12 节。虽然本节示例主要针对 UNIX，但本节讨论中关于从 zip 文件导入模块的信息和内容在 Windows 下同样有效。

更多资料

Library Reference 和 *Python in a Nutshell* 中 zipfile，tempfile，os，sys 模块的资料；关于归档文件树的内容，参见第 2.11 节；更多有关从 zip 文件导入模块的信息，参见第 16.12 节。

2.10 处理字符串中的 zip 文件

感谢：Indyana Jones

任务

你的程序接收到了一个字符串，其内容是一个 zip 文件，需要读取这个 zip 文件中的信息。

解决方案

应对这种问题，正是 Python 标准库的 cStringIO 模块的拿手好戏：

```
import cStringIO, zipfile
class ZipString(ZipFile):
     def __init__(self, datastring):
          ZipFile.__init__(self, cStringIO.StringIO(datastring))
```

讨论

我总是遇到这类任务——比如 zip 文件可能来自于数据库的 BLOB 字段，或者来自于网络连接。我过去把这些二进制数据先存成临时文件，然后用标准库模块 zipfile 打开此文件。当然，我会确保完成任务之后删除临时文件。一天，我想到了使用标准库模块 cStringI，之后我就再也没用过老办法了。

cStringIO 模块可以将一串字节封装起来，让你像访问文件对象一样访问其中的数据。另一方面，还可以用把 cStringIO.StringIO 的实例当做一个文件对象，向其中写入数据，最后得到的是一串内存中的字节。很多处理文件的 Python 模块其实根本不检查你传递给它们的是不是一个真正的文件——任何像文件一样的对象它们都接受。它们只是在需要的时候调用文件的方法，只要给这些对象提供了相关的方法，并且做出了正确的反应，一切都会正常工作。这展示了基于签名的多态机制的强大力量，同时也解释了为什么最好不要在你的代码中进行类型检查（比如可怕的 if type(x) is y，以及稍微好点的 if isinstance(x, y)）。一些低级的模块，比如 marshal，很不幸地会执着要求“真实”的文件，但 zipfile 很通融，本节的例子也说明了，有了它生活多么美好。

如果用的 Python 版本和主流的基于 C 的 Python 版本（也称为 CPython）不同，可能在标准库中找不到 cStringIO 模块。模块名最前面的 c，表示它是个基于 C 的模块，为速度做过优化，不保证能够在其他兼容的 Python 实现体的标准库中找到对应的版本。这些兼容的 Python 实现体包括了产品级品质的（比如 Jython，用 Java 实现并运行在 JVM 上）以及实验性的（比如 pypy，用 Python 代码产生机器码结果，以及 IronPython，基于 C#并运行在微软的.NET CLR）。别担心，Python 标准库总会包含 StringIO 模块，它由纯 Python 代码实现（因此也能适用于任何兼容的 Python 实现体），并实现了和 cStringIO 一样的功能（不过没有那么快，至少在主流的 CPython 上是这样）。只需略微修改一下 import 语句，以确保如果有 cStringIO，就导入 cStringIO，如果没有，则导入 StringIO 作为代替。比如，代码可能会变成这样：

```
import zipfile
try:
     from cStringIO import StringIO
except ImportError:
     from StringIO import StringIO
class ZipString(ZipFile):
     def __init__(self, datastring):
          ZipFile.__init__(self, StringIO(datastring))
```

经过这次修改，这段代码就可以在 Jython 或其他的实现体中工作了。

更多资料

Library Reference 和 *Python in a Nutshell* 中的 zipfile 和 cStringIO 模块；Jython 的信息参见 http://www.jython.org/；pypy 的信息参见 http://codespeak.net/pypy/；IronPython 的信息参见 http://ironpython.com/。

2.11 将文件树归档到一个压缩的 tar 文件

感谢：Ed Gordon、Ravi Teja Bhupatiraju

任务

需要将一个文件树中的所有文件和子目录归档到一个 tar 归档文件，然后用流行的 gzip 方式或者更高压缩率的 bzip2 方式来压缩。

解决方案

Python 标准库的 tarfile 模块直接提供了这两种压缩方式，你只需在调用 tarfile.TarFile.open 创建归档文件时，传入一个选项字符串以指定需要的压缩方式即可。比如：

```
import tarfile, os
def make_tar(folder_to_backup, dest_folder, compression='bz2'):
     if compression:
          dest_ext = '.' + compression
     else:
          dest_ext = ''
     arcname = os.path.basename(folder_to_backup)
     dest_name = '%s.tar%s' % (arcname, dest_ext)
     dest_path = os.path.join(dest_folder, dest_name)
     if compression:
          dest_cmp = ':' + compression
     else:
          dest_cmp = ''
     out = tarfile.TarFile.open(dest_path, 'w'+dest_cmp)
     out.add(folder_to_backup, arcname)
     out.close( )
     return dest_path
```

讨论

可以给函数 make_tar 传递一个指定压缩方式的参数，字符串“gz”则表明使用 gzip 压缩，默认的则是“bz2”，指定 bzip2 压缩。也可以传递一个空字符串”，表明你压根就不需要压缩。当你选择不压缩、用 gzip 压缩或者 bzip2 压缩时，除了会影响到文件扩展名成为.tar、.tar.gz 或.tar.bz2 之外，你的选择还决定了“w”，“w:gz”和“w:bz2”中

哪个字符串会被作为第二个参数传递给 tarfile.TarFile.open。

除了 open 之外，类 tarfile.TarFile 提供了几种其他的类方法（classmethods），可以用这些方法生成一个合适的实例。我发现 open 最方便灵活，因为它可以从模式字符串参数中获取指定压缩方式的信息。当然，如果确定无条件地使用 bzip2 压缩方式，也可以用类方法 bz2open 来替代 open。

一旦我们拥有了 tarfile.TarFile 的一个实例，并且也设置好了我们需要的压缩方式，这个实例的 add 方法将完成所有剩下的工作。特别是，当字符串 folder_to_backup 是“目录”而不是普通文件的名字时，add 会递归地把该目录中所有的子树添加进来。在另一些场合中，我们可能会希望改变默认的行为，并精确地控制需要被归档的文件和目录，我们可以给 add 传递一个额外的参数 recursive=False 来关闭默认的递归添加功能。在调用 add 之后，留给 make_tar 函数做的事情就是关闭 TarFile 实例并返回所写入的 tar 文件路径，这是因为有时调用者会需要使用这个信息。

更多资料

Library Reference 中关于 tarfile 模块的文档。

2.12 将二进制数据发送到 Windows 的标准输出

感谢：Hamish Lawson

任务

在 Windows 平台上，你想把二进制数据（比如一张图片）发送到 stdout 中。

解决方案

Python 标准库中，依赖特定平台（Windows）的模块 msvcrt 提供了 setmode 函数，可用来完成这个任务：

```
import sys
if sys.platform == "win32":
      import os, msvcrt
      msvcrt.setmode(sys.stdout.fileno( ), os.O_BINARY)
```

现在可以给 sys.stdout.write 任何字节或者字符串参数，这些字节和字符串会被不加修改地传递到标准输出中。

讨论

由于 UNIX 并不（或不需要）区分文本和二进制模式，如果打算在 Windows 中读取或者写入二进制数据，比如图片，则必须以二进制模式打开文件。这对于向标准输出（比

如，CGI 脚本就可能会这么干）写入二进制数据的程序而言是个问题，因为 Python 通常以文本模式打开 sys.stdout 文件对象。

可以指定命令行选项-u 以二进制模式打开 stdout。比如，如果你知道你的 CGI 脚本运行在 Apache web 服务器中，可以这样写你的脚本的第一行：

```
#! c:/python23/python.exe -u
```

这假设了用的是 Python 2.3 的标准安装。不过，并不总能控制你的脚本在什么样的命令行中运行。解决方案中给出的是一个可行的选择。setmode 函数提供了 Windows 专用的 msvcrt 模块，以方便用户修改 stdout 固有的文件描述符。通过这个函数，可以在程序内部确认 sys.stdout 被设置为二进制模式。

更多资料

Library Reference 和 *Python in a Nutshell* 中 msvcrt 模块的文档。

2.13 使用 C++的类 iostream 语法

感谢：Erik Max Francis

任务

你喜爱 C++的基于 ostream 和操纵符（插入了这种特定的对象后，它会在 stream 中产生特定的效果）的 I/O 方式，并想将此形式用在自己的 Python 程序中。

解决方案

Python 允许使用对特殊方法（即名字前后带有连续两个下划线的方法）进行了重定义的类来重载原有的操作符。为了将<<用于输出，如同在 C++中所做的一样，需要编写一个输出流类，并定义特殊方法__lshift__：

```
class IOManipulator(object):
     def __init__(self, function=None):
          self.function = function
     def do(self, output):
          self.function(output)
def do_endl(stream):
     stream.output.write('\n')
     stream.output.flush( )
endl = IOManipulator(do_endl)
class OStream(object):
     def __init__(self, output=None):
          if output is None:
               import sys
               output = sys.stdout
```

```
            self.output = output
            self.format = '%s'
        def __lshift__(self, thing):
            ''' 当你使用<<操纵符并且左边操作对象是 OStream 时,
                Python 会调用这个特殊方法 '''
            if isinstance(thing, IOManipulator):
                thing.do(self)
            else:
                self.output.write(self.format % thing)
                self.format = '%s'
            return self
    def example_main( ):
        cout = OStream( )
        cout<< "The average of " << 1 << " and " << 3 << " is " << (1+3)/2 <<endl
    # 输出: The average of 1 and 3 is 2
    if __name__ == '__main__':
        example_main( )
```

讨论

在 Python 中包装一个像文件一样的对象，模拟 C++的 ostream 的语法，还算比较容易。本节展示了怎样编写代码实现插入操作符<<的效果。解决方案中的代码实现了一个 IOManipulator 类（像 C++中一样）来调用插入到流中的任意函数，还实现了预定义的操纵符 endl（猜猜它得名何处）来写入新行和刷新流。

Ostream 类的实例有一个叫做 format 的属性，在每次调用 self.output.write 之后，这个属性都会被设置为默认值“%s”，这样的好处是，每次创建一个操纵符之后我们可在其中临时保存流对象的格式化状态，比如：

```
def do_hex(stream):
    stream.format = '%x'
hex = IOManipulator(do_hex)
cout << 23 << ' in hex is ' << hex << 23 << ', and in decimal ' << 23 << endl
# 输出: 23 in hex is 17, and in decimal 23
```

一些人很讨厌 C++的 cout<<something 的语法，另一些人却很喜欢。在本节的例子中，所用的语法至少在可读性和简洁方面都胜过下面的语法：

```
print>>somewhere, "The average of %d and %d is %f\n" % (1, 3, (1+3)/2)
```

这种方式是 Python“原生”的方式（看上去很像 C 的风格）。这要看你更习惯 C++还是 C，至少本节给了你另一个选择。即使最终没有使用本节提供的方式，了解 Python 中简单的操作符重载还是蛮有趣的。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于文件对象以及特殊方法_ _lshift_ _的文档；第 4.20 节中关于 C 函数 printf 的 Python 实现的信息。

2.14 回退输入文件到起点

感谢：Andrew Dalke

任务

需要创建一个输入文件对象（数据可能来自于网络 socket 或者其他输入文件句柄），此文件对象允许回退到起点，这样就可以完全读取其中所有数据。

解决方案

将文件对象封装到一个合适的类中：

```
from cStringIO import StringIO
class RewindableFile(object):
     """ 封装一个文件句柄以便重定位到开始位置 """
    def __init__(self, input_file):
          """ 将 input_file 封装到一个支持回退的类文件对象中 """
          self.file = input_file
          self.buffer_file = StringIO( )
          self.at_start = True
          try:
               self.start = input_file.tell( )
          except (IOError, AttributeError):
               self.start = 0
          self._use_buffer = True
     def seek(self, offset, whence=0):
          """ 根据给定的字节定位.
          必须:  whence == 0 and offset == self.start
          """
          if whence != 0:
               raise ValueError("whence=%r; expecting 0" % (whence,))
          if offset != self.start:
               raise ValueError("offset=%r; expecting %s" % (offset,
               self.start))
          self.rewind( )
     def rewind(self):
          """ 回到起始位置"""
          self.buffer_file.seek(0)
          self.at_start = True
     def tell(self):
          """ 返回文件的当前位置（必须在开始处）"""
          if not self.at_start:
               raise TypeError("RewindableFile can't tell except at start
               of file")
          return self.start
     def _read(self, size):
```

```
        if size < 0:            # 一直读到文件末尾
            y = self.file.read( )
            if self._use_buffer:
                self.buffer_file.write(y)
            return self.buffer_file.read( ) + y
        elif size == 0:         # 不必读空字符串
            return ""
        x = self.buffer_file.read(size)
        if len(x) < size:
            y = self.file.read(size - len(x))
            if self._use_buffer:
                self.buffer_file.write(y)
            return x + y
        return x
    def read(self, size=-1):
        """ 根据 size 指定的大小读取数据
        默认为-1，意味着一直读到文件结束
        """
        x = self._read(size)
        if self.at_start and x:
            self.at_start = False
        self._check_no_buffer( )
        return x
    def readline(self):
        """ 从文件中读取一行"""
        # buffer_file 中有吗?
        s = self.buffer_file.readline( )
        if s[-1:] == "\n":
            return s
        # 没有，从输入文件中读取一行
        t = self.file.readline( )
        if self._use_buffer:
            self.buffer_file.write(t)
        self._check_no_buffer( )
        return s + t
    def readlines(self):
        """读取文件中所有剩余的行"""
        return self.read( ).splitlines(True)
def _check_no_buffer(self):
        # 如果'nobuffer'被调用，而且我们也完成了对缓存文件的处理
        # 那就删掉缓存，把所有的东西都重定向到原来的输入文件
        if not self._use_buffer and \
                self.buffer_file.tell()==len(self.buffer_file.
                getvalue()):
            # 为了获得尽可能高的性能，我们重新绑定了 self 中的所有相关方法
            for n in 'seek tell read readline readlines'.split( ):
                setattr(self, n, getattr(self.file, n, None))
            del self.buffer_file
    def nobuffer(self):
        """通知 RewindableFile，一旦缓存耗尽就停止继续使用缓存"""
        self._use_buffer = False
```

讨论

有时，从 socket 或其他输入文件句柄中得来的数据并不是我们想要的。比如，假设从一个有问题的服务器读取数据，此服务器应该返回 XML 流，但它有时却给你未格式化的错误信息。（这种情况时常发生，因为很多服务器并不能正确处理错误输入。）

本节的 RewindableFile 类能够帮助你解决此类问题。r = RewindableFile(f)将原来的输入流 f 封装进了一个“可回退的文件”的实例 r，r 模仿 f 的行为，但同时提供了缓存。对 r 的读取请求被转移到了 f，读取的数据则添加进了缓存，然后返回给调用者。缓存中保存着到目前为止读取的所有数据。

r 可以回退，也即定位到开始位置。下一个请求读取的内容可能会来自于缓存，直到缓存被全部读取，此时它会从输入流中获取数据。新读取的数据也被添加到缓存中。

如果不再需要缓存了，可直接调用 r 的 nobuffer 方法。这相当于告诉 r，一旦它读完了缓存中的当前内容，它就可以丢弃缓存。调用 nobuffer 之后，seek 的行为就处于未定义状态了。

举个例子，假设有个服务器，它可能会给你错误信息，其形式为“ERROR: cannot do that”，或者给你一个 XML 数据流，其内容以“<?xml…”开始：

```
import RewindableFile
infile = urllib2.urlopen("http://somewhere/")
infile = RewindableFile.RewindableFile(infile)
s = infile.readline( )
if s.startswith("ERROR:"):
     raise Exception(s[:-1])
infile.seek(0)
infile.nobuffer( )     # 不再缓存数据
     ...process the XML from infile...
```

一些在某些场合下很有用的 Python 惯用方式在此类中不被支持：无法可靠地将 RewindableFile 实例的被绑定方法（bound method）隐藏起来（如果不知道什么叫做被绑定方法，没关系，反正在这里肯定无法把它们藏在任何地方）。出现这种限制的原因是，当缓存空了，RewindableFile 代码会给输入文件的 read、readlines 等方法重新赋值，作为 self 实例的变量。这样会有略好的性能，代价是不再支持常用的那种保存被绑定方法的惯用方式。6.11 节中会给出另一个相似的技术，不过其中的实例不可逆转地改变了它自己的方法。

获取当前位置的 tell 方法，只能正好在完成了封装之后，在读取任何数据之前，由 RewindableFile 实例调用，以获取起始字节位置。RewindableFile 实现了 tell 方法，是为了得到被封装的文件对象的真实位置，并以此为起始位置。如果被封装的文件不支持 tell，那么 RewindableFile 实现的 tell 会返回 0。

更多资料

在 http://www.dalkescientific.com/Python/有本节代码的最新版本；*Library Reference* 和 *Python in a Nutshell* 中关于文件对象和 cStringIO 模块的内容；参考第 6.11 节中另一个通过重绑定不可逆地改变自身行为的例子。

2.15 用类文件对象适配真实文件对象

感谢：Michael Kent

任务

需要传递一个类似文件的对象（比如，调用 urllib.urlopen 返回的结果）给一个函数或者方法，但这个函数或方法要求只接受真实的文件对象（比如，像 marshal.load 这样的函数）。

解决方案

为了过类型检查这一关，我们需要将类文件对象中的所有数据写入到磁盘中的一个临时文件。然后使用临时文件的（真实）文件对象。下面给出一个实现这个想法的函数：

```
import types, tempfile
CHUNK_SIZE = 16 * 1024
def adapt_file(fileObj):
     if isinstance(fileObj, file): return fileObj
     tmpFileObj = tempfile.TemporaryFile
     while True:
          data = fileObj.read(CHUNK_SIZE)
          if not data: break
          tmpFileObj.write(data)
     fileObj.close( )
     tmpFileObj.seek(0)
     return tmpFileObj
```

讨论

本节展示的其实是设计模式中的适配器（即 Adapter，比如你想要 X，我却给你 Y 以替换 X）的 Python 风格的实现。虽然设计模式通常被认为是一种面向对象的设计方式，因此一般需要用类来实现，但具体实现并没有什么限制。比如此例中，我们根本不需要引入任何新类，因为 adapt_file 函数已经足够了。这里我们遵守奥卡姆剃刀原理（译者注：Occam's Razor，奥卡姆是 14 世纪的一个逻辑学家和天主教修道士，奥卡姆剃刀原理即“如无必要，勿增实体”），不在没有必要的情况下引入任何实体。

当需要依赖一些底层的、要求精确类型的工具时，应该首先考虑适配，而不是类型检查。当获得一个适合的可以绕过类型检查的对象时，应该考虑将其配接成需要的对象。用这种方式，你的代码会更加灵活，也更具复用性。

更多资料

Library Reference 和 *Python in a Nutshell* 中内建文件对象、tempfile 和 marshal 模块的内容。

2.16 遍历目录树

感谢：Robin Parmar、Alex Martelli

任务

需要检查一个“目录”，或者某个包含子目录的目录树，并根据某种模式迭代所有的文件（也可能包含子目录）。

解决方案

Python 标准库模块 os 中的生成器（generator）os.walk 对于这个任务来说完全够用了，不过我们可以给它打扮打扮，将其封装为一个我们自己的函数：

```
import os, fnmatch
def all_files(root, patterns='*', single_level=False, yield_folders=False):
     # 将模式从字符串中取出放入列表中
     patterns = patterns.split(';')
     for path, subdirs, files in os.walk(root):
          if yield_folders:
               files.extend(subdirs)
          files.sort( )
          for name in files:
               for pattern in patterns:
                    if fnmatch.fnmatch(name, pattern):
                         yield os.path.join(path, name)
                         break
          if single_level:
               break
```

讨论

标准文件树遍历生成器 os.walk 既强大又简单灵活。不过，os.walk 还缺乏应用程序需要的一些细节上的处理能力，比如根据某种模式选择文件，扁平（线性）地以排序后的顺序循环所有文件（也可能包括子目录），检查一个单一目录（不进入其子目录）。本节代码则展示了，通过将 os.walk 封装到另一个简单的生成器中，并使用标准库模块

fnmatch 来检查文件名匹配模式，是多么的简单方便。

文件名匹配模式可能是大小写无关的（这依赖于平台），也可能是相关的，比如 UNIX 风格，不过这些能力都是标准 fnmatch 模块能够提供的。为了指定多个模式，可用分号将它们连接起来。注意，这意味着那些分号本身不是模式的一部分。

举个例子，可以很容易地从/tmp 目录及其子目录中获得一个包括所有 Python 和 HTML 文件的列表：

```
thefiles = list(all_files('/tmp', '*.py;*.htm;*.html'))
```

如果想一次处理一个文件的路径（比如，逐行打印它们），不需要先建立一个列表：可以直接对 all_files 调用的结果进行循环操作：

```
for path in all_files('/tmp', '*.py;*.htm;*.html'):
    print path
```

如果你的平台是大小写敏感的，而且你也希望严格匹配大小写，那么需要在指定模式的时候稍微辛苦点，比如，用“*.[Hh][Tt][Mm][Ll]”来替代原来的“*.html”。

更多资料

参看 *Library Reference* 和 *Python in a Nutshell* 中的 os.path，fnmatch 模块，以及 os.walk 生成器。

2.17 在目录树中改变文件扩展名

感谢：Julius Welby

任务

需要在一个目录的子树中重命名一系列文件，具体地说，你想将某一指定类型的文件的扩展名改成另一种扩展名。

解决方案

用 Python 标准库提供的 os.walk 函数来处理子目录中的所有文件，任务变得非常容易：

```
import os
def swapextensions(dir, before, after):
    if before[:1] != '.':
        before = '.'+before
    thelen = -len(before)
    if after[:1] != '.':
        after = '.'+after
    for path, subdirs, files in os.walk(dir):
        for oldfile in files:
```

```
                if oldfile[thelen:] == before:
                    oldfile = os.path.join(path, oldfile)
                    newfile = oldfile[:thelen] + after
                    os.rename(oldfile, newfile)
if __name__=='__main__':
    import sys
    if len(sys.argv) != 4:
        print "Usage: swapext rootdir before after"
        sys.exit(100)
    swapextensions(sys.argv[1], sys.argv[2], sys.argv[3])
```

讨论

本节展示了怎样改变一个指定目录中所有文件的扩展名，涉及范围包括了所有的子目录，以及更下级子目录，以此类推。这种技术很适合在一个文件夹结构中批量修改文件的扩展名，比如针对一个 web 站点的目录树进行修改。可以用这个脚本纠正用程序批量生成文件时所犯的错误。

本节给的代码既可以被用作一个可以随时导入的模块，也可以作为一个脚本并运行在命令行中，而且代码设计得很谨慎，完全是平台无关的。可以传入带点（.）的扩展名，也可以传入不带点的，程序在必要时会自行插入点。（作为这种方便性的一个直接后果是，此程序不能处理没有扩展名的文件，也不能直接处理点，在 UNIX 系统中这种限制有时很让人恼火。）

实现本节解决方案所用的技术，一些完美主义者会认为过于底层——直接用操作字符串的方式来修改处理文件名和扩展名，而不是用 os.path 提供的函数。不过这没什么大不了的：用 os.path 很好，但是用 Python 的强大的字符串工具也很好。

更多资料

参看作者的主页 http://www.outwardlynormal.com/python/swapextensions.htm。

2.18 从指定的搜索路径寻找文件

感谢：Chui Tey

任务

给定一个搜索路径（一个描述目录信息的字符串），需要根据这个路径和请求的文件名找到第一个符合要求的文件。

解决方案

基本上，需要循环指定的搜索路径中的目录：

```
import os
def search_file(filename, search_path, pathsep=os.pathsep):
     """ 给定一个搜索路径，根据请求的名字找到文件 """
     for path in search_path.split(pathsep):
          candidate = os.path.join(path, filename)
          if os.path.isfile(candidate):
               return os.path.abspath(candidate)
     return None
if __name__ == '__main__':
     search_path = '/bin' + os.pathsep + '/usr/bin'  # ; on Windows, : on UNIX
     find_file = search_file('ls', search_path)
     if find_file:
          print "File 'ls' found at %s" % find_file
     else:
          print "File 'ls' not found"
```

讨论

本节的任务是个很常见的需求，Python 对这个需求的解决办法也极其简单。本章其他一些节也会处理相似或相关的一些任务：见第 2.20 节，在 Python 自身的搜索路径中找文件，以及第 2.19 节，在指定的搜索路径中根据匹配模式寻找文件。

进行搜索的循环可以被写成很多形式，但一旦找到就立刻返回路径（这里用绝对路径，主要基于统一性和方便性的考虑）是最简单的，而且速度很快。在循环完成之后显式地 return None 并不是必须的，因为在 Python 中一个函数执行完毕后会自行返回 None。在这里画蛇添足的加一句 return 语句，仅仅是为了让人能够一目了然地看清 search_file 的所做的事情。

更多资料

2.20；2.10；*Library Reference* 和 *Python in a Nutshell* 中 os 模块的内容。

2.19 根据指定的搜索路径和模式寻找文件

感谢：Bill McNeill、Andrew Kirkpatrick

任务

给定一个搜索路径（一个描述目录信息的字符串），需要在此目录中找出所有符合匹配模式的文件。

解决方案

基本上，需要循环路径中的所有目录。这个循环最好被封装成一个生成器：

```
import glob, os
def all_files(pattern, search_path, pathsep=os.pathsep):
     """ 给定搜索路径，找出所有满足匹配条件的文件 """
     for path in search_path.split(pathsep):
          for match in glob.glob(os.path.join(path, pattern)):
               yield match
```

讨论

生成器的好处是，可以很容易地获取第一个子项，或者所有子项，再或者其中任意一个子项。比如，打印出你的环境变量 PATH 中第一个符合“*.pye”模式的文件：

```
print all_files('*.pye', os.environ['PATH']).next( )
```

打印所有这种文件，一行一个：

```
for match in all_files('*.pye', os.environ['PATH']):
     print match
```

以列表形式一次全部打印出来：

```
print list(all_files('*.pye', os.environ['PATH']))
```

我也给 all_files 函数提供了一个主脚本，以方便打印出我的 PATH 中所有符合匹配模式的文件。因此，不仅能够看到根据指定名字将被执行的那个文件（第一个），还能看到被第一个文件“屏蔽”掉的其他同名文件：

```
if __name__ == '__main__':
     import sys
     if len(sys.argv) != 2 or sys.argv[1].startswith('-'):
          print 'Use: %s <pattern>' % sys.argv[0]
          sys.exit(1)
     matches = list(all_files(sys.argv[1], os.environ['PATH']))
     print '%d match:' % len(matches)
     for match in matches:
          print match
```

更多资料

见第 2.18 节中的那个更简单的根据搜索路径寻找文件的例子；*Library Reference* 和 *Python in a Nutshell* 中的 os 和 glob 的相关文档。

2.20 在 Python 的搜索路径中寻找文件

感谢：Mitch Chapman

任务

一个大的 Python 应用程序包括了资源文件（比如 Glade 项目文件、SQL 模板和图片）

以及 Python 包（Python package）。你想把所有这些相关文件和用到它们的 Python 包储存起来。

解决方案

可以在 Python 的 sys.path 中寻找文件或目录：

```
import sys, os
class Error(Exception): pass
def _find(pathname, matchFunc=os.path.isfile):
     for dirname in sys.path:
          candidate = os.path.join(dirname, pathname)
          if matchFunc(candidate):
               return candidate
     raise Error("Can't find file %s" % pathname)
def findFile(pathname):
     return _find(pathname)
def findDir(path):
     return _find(path, matchFunc=os.path.isdir)
```

讨论

比较大的 Python 应用程序由一系列 Python 包和相关的资源文件组成。将这些相关文件和用到它们的 Python 包一起储存起来是很方便的，可以很容易地对 2.18 提供的代码略加修改，使之能根据 Python 搜索路径的相对路径来寻找文件和目录。

更多资料

2.18 节；*Library Reference* 和 *Python in a Nutshell* 中的 os 模块相关内容。

2.21 动态地改变 Python 搜索路径

感谢：Robin Parmar

任务

模块必须处于 Python 搜索路径中才能被导入，但你不想设置个永久性的大路径，因为那样可能会影响性能，所以，你希望能够动态地改变这个路径。

解决方案

只需简单地在 Python 的 sys.path 中加入一个“目录”，不过要小心重复的情况：

```
def AddSysPath(new_path):
     """ AddSysPath(new_path): 给 Python 的 sys.path 增加一个“目录”
     如果此目录不存在或者已经在 sys.path 中了，则不操作
```

```
        返回 1 表示成功，-1 表示 new_path 不存在，0 表示已经在 sys.path 中了
        already on sys.path.
        """
        import sys, os
        # 避免加入一个不存在的目录
        if not os.path.exists(new_path): return -1
        # 将路径标准化。 Windows 是大小写不敏感的，所以若确定在
        # Windows 下，将其转成小写
        new_path = os.path.abspath(new_path)
        if sys.platform == 'win32':
            new_path = new_path.lower( )
        # 检查当前所有的路径
        for x in sys.path:
            x = os.path.abspath(x)
            if sys.platform == 'win32':
                x = x.lower( )
            if new_path in (x, x + os.sep):
                return 0
        sys.path.append(new_path)
        # 如果想让 new_path 在 sys.path 处于最前
        # 使用: sys.path.insert(0, new_path)
        return 1
if __name__ == '__main__':
        # 测试，显示用法
        import sys
        print 'Before:'
        for x in sys.path: print x
        if sys.platform == 'win32':
            print AddSysPath('c:\\Temp')
            print AddSysPath('c:\\temp')
    else:
            print AddSysPath('/usr/lib/my_modules')
        print 'After:'
        for x in sys.path: print x
```

讨论

模块要处于 Python 搜索路径中的目录里才能被导入，但我们不喜欢维护一个永久性的大目录，因为其他所有的 Python 脚本和应用程序导入模块的时候性能都会被拖累。本节代码动态地在该路径中添加了一个“目录”，当然前提是此目录存在而且此前不在 sys.path 中。

sys.path 是个列表，所以在末尾添加目录是很容易的，用 sys.path.append 就行了。当这个 append 执行完之后，新目录即时起效，以后的每次 import 操作都可能会检查这个目录。如同解决方案所示，可以选择用 sys.path.insert(0,…，这样新添加的目录会优先于其他目录被 import 检查。

即使 sys.path 中存在重复，或者一个不存在的目录被不小心添加进来，也没什么大不了，Python 的 import 语句非常聪明，它会自己应付这类问题。但是，如果每次 import 时都发生这种错误（比如，重复的不成功搜索，操作系统提示的需要进一步处理的错误），我们会被迫付出一点小小的性能代价。为了避免这种无谓的开销，本节代码在向 sys.path 添加内容时非常谨慎，绝不加入不存在的目录或者重复的目录。程序向 sys.path 添加的目录只会在此程序的生命周期之内有效，其他所有的对 sys.path 的动态操作也是如此。

更多资料

Library Reference 和 *Python in a Nutshell* 中 sys 和 os.path 模块的内容。

2.22 计算目录间的相对路径

感谢：Cimarron Taylor、Alan Ezust

任务

需要知道一个目录对另一个目录的相对路径是什么——比如，有时需要创建一个符号链接或者一个相对的 URL 引用。

解决方案

最简单的方法是把目录拆分到一个目录的列表中，然后对列表进行处理。我们需要用到一些辅助函数和助手函数，代码如下：

```
import os, itertools
def all_equal(elements):
    ''' 若所有元素都相等，则返回 True，否则返回 False'''
    first_element = elements[0]
    for other_element in elements[1:]:
        if other_element != first_element: return False
    return True
def common_prefix(*sequences):
    ''' 返回所有序列开头部分共同元素的列表
        紧接一个各序列的不同尾部的列表'''
    # 如果没有 sequence，完成
    if not sequences: return [ ], [ ]
    # 并行地循环序列
    common = [ ]
    for elements in itertools.izip(*sequences):
        # 若所有元素相等，跳出循环
        if not all_equal(elements): break
        # 得到一个共同的元素，添加到末尾并继续
        common.append(elements[0])
```

```
        # 返回相同的头部和各自不同的尾部
        return common, [ sequence[len(common):] for sequence in sequences ]
    def relpath(p1, p2, sep=os.path.sep, pardir=os.path.pardir):
        ''' 返回 p1 对 p2 的相对路径
            特殊情况：空串，if p1 == p2;
                      p2，如果 p2 和 p1 完全没有相同的元素
        '''
        common, (u1, u2) = common_prefix(p1.split(sep), p2.split(sep))
        if not common:
            return p2       # 如果完全没有共同元素，则路径是绝对路径
        return sep.join( [pardir]*len(u1) + u2 )
    def test(p1, p2, sep=os.path.sep):
        ''' 调用 relpath 函数，打印调用参数和结果 '''
        print "from", p1, "to", p2, " -> ", relpath(p1, p2, sep)
    if __name__ == '__main__':
        test('/a/b/c/d', '/a/b/c1/d1', '/')
        test('/a/b/c/d', '/a/b/c/d', '/')
        test('c:/x/y/z', 'd:/x/y/z', '/')
```

讨论

本节解决方案给出的代码中，简单而通用的 common_prefix 是关键部分，给它任意 N 个序列，它能返回 N 个序列共同的头部，和一个各不相同的尾部列表。为了计算两个目录之间的相对路径，可以忽略掉它们的共同头部。我们只需要一定数目的“向上一级”标记（通常用 os.path.pardir，比如类 UNIX 系统中的../；需要和尾部长度相同数目的这种符号），然后再添上目标目录的尾部。relpath 函数将一个完整路径拆成一个目录的列表，然后调用 common_prefix，接着执行我们刚才描述过的操作。

common_prefix 的核心部分在那个循环，for elements in itertools.izip(*sequences)，它依赖这个事实：当最短的序列循环到头时，izip 也结束了。循环的主体部分必须在遇到一个不全相等的元组（根据 izip 的说明，每个元素都来自各序列）时立刻结束，同时还要在这个过程中把全等的元素放进 common 列表中保存起来。一旦循环结束，剩下要做的事情就是根据 common 列表，把各个序列的相同头部全部切掉。

all_equal 函数还能用另一种方式实现，没那么简洁明快，但是也很有趣：

```
def all_equal(elements):
    return len(dict.fromkeys(elements)) == 1
```

或者，等价但更简洁一点，适用于 Python 2.4 以上：

```
def all_equal(elements):
    return len(set(elements)) == 1
```

所有元素相等，等价于包含且只包含这些元素的集合的势（cardinality）为 1。在使用 dict.fromkeys 的变体中，用 dict 来代替 set，所以那个例子可同时适用于 Python 2.3 和 2.4。用 set 的那个例子更清晰，但只能在 Python 2.4 以上的版本中使用（可以用 Python

标准库中的 sets 模块来改写，让它也同时适用于 Python 2.3）。

更多资料

Library Reference 和 *Python in a Nutshell* 中的 os 和 itertools 模块。

2.23 跨平台地读取无缓存的字符

感谢：Danny Yoo

任务

你的程序需要从标准输出中，读取无缓存的单个的字符，而且它还必须能够同时在 Windows 和类 UNIX 系统中工作。

解决方案

当我们手上的工具是平台依赖性的，而需求却是平台无关的时候，可以试试将这些差异封装起来：

```
try:
    from msvcrt import getch
except ImportError:
     ''' 我们不在 Windows 中，所以可以尝试类 UNIX 的方式 '''
     def getch( ):
          import sys, tty, termios
          fd = sys.stdin.fileno( )
          old_settings = termios.tcgetattr(fd)
          try:
               tty.setraw(fd)
               ch = sys.stdin.read(1)
          finally:
               termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
          return ch
```

讨论

在 Windows 中，Python 标准库模块 msvcrt 提供了方便的 getch 函数来读取无缓存的单个字符，直接从键盘读取，且不会在屏幕上回显。但是这个模块并不是 UNIX 和类 UNIX 平台中 Python 标准库的一部分，比如 Linux 和 Mac OS X 的 Python 标准库中就不提供这个模块。在这些平台中，只能利用 Python 标准库的 tty 和 termios 模块（同样的，这些模块在 Windows 平台中也未提供对应功能）来实现类似的功能。

在应用级的程序代码中，我们几乎从来不应该考虑这种问题；我们更倾向于将程序写得更加平台无关，并依赖库函数来实现在不同系统间的跨越。Python 标准库对于大多

数任务的跨平台能力都提供了极其优秀的支持，但本节任务中的需求，却正好是 Python 标准库没有提供相应的跨平台解决方案的一个例子。

当我们不能在标准库中找到可用的跨平台包或工具时，应该自己设法打包，并作为自己的附加自定义库的一部分。本节的解决方案，不仅解决了一个特别的任务，同时还展示了一种好的通用封装方式。（也可以选择测试 sys.platform，但我更喜欢本节给出的方法。）

你自己的库模块在特定系统下导入标准库模块时，应该尝试用 Try 语句，包括其对应的 except ImportError 语句，当运行的系统不符合要求时 except ImportError 会被激发。在 except 语句中，你的库模块可以选择任何其他可以在当前系统工作的方法。在一些比较少见的情况下，你可能会需要两种以上的平台相关的方法，但绝大多数情况下，你都只需准备一个在 Windows 下工作的方法和另一个适用于所有其他平台的方法。这是因为当今的绝大多数非 Windows 平台基本上都是 UNIX 或者类 UNIX。

更多资料

Library Reference 和 *Python in a Nutshell* 中 msvcrt，tty 和 termios 的内容。

2.24 在 Mac OS X 平台上统计 PDF 文档的页数

感谢：Dinu Gherman、Dan Wolfe

任务

你的计算机运行着比较新的 Mac OS X 系统（10.3 的“Panther”或更新的版本），现在需要知道一个 PDF 文档的页数。

解决方案

PDF 格式和 Python 都已经集成到了 Mac OS X 系统中（10.3 或更高版本），因而这个问题解决起来也相对比较容易：

```
#!/usr/bin python
import CoreGraphics
def pageCount(pdfPath):
     " 返回指定路径的 PDF 文档的页数 "
     pdf = CoreGraphics.CGPDFDocumentCreateWithProvider(
               CoreGraphics.CGDataProviderCreateWithFilename(pdfPath)
          )
     return pdf.getNumberOfPages( )
if __name__ == '__main__':
     import sys
     for path in sys.argv[1:]:
          print pageCount(path)
```

讨论

另一个完成任务的方法是使用 Python 扩展，PyObjC，它使得 Python 代码可以利用 Mac OS X 所带的 Foundation 和 AppKit 框架的能力。该方案也可以让你的代码运行在较老版本的 Mac OS X 中，比如 10.2 Jaguar。不过依赖并使用 Mac OS X 10.3 或更高版本提供的集成 Python 的环境和 CoreGraphics 扩展（也是 Mac OS X “Panther”的一部分），可以使我们的代码可直接利用 Apple 强大的 Quartz 图形引擎。

更多资料

关于 PyObjC，参看 http://pyobjc.sourceforge.net/；更多有关 CoreGraphics 模块的资料则请访问 http://www.macdevcenter.com/pub/a/mac/2004/03/19/core_graphics.html。

2.25 在 Windows 平台上修改文件属性

感谢：John Nielsen

任务

需要修改 Windows 上一系列文件的属性，比如将某些文件设置为只读、归档等。

解决方案

PyWin32 的 win32api 模块提供了一个 SetFileAttributes 函数，正好可以用来完成这种任务：

```
import win32con, win32api, os
# 创建一个文件，并展示如何操纵它
thefile = 'test'
f = open('test', 'w')
f.close( )
# 设置成隐藏文件...:
win32api.SetFileAttributes(thefile, win32con.FILE_ATTRIBUTE_HIDDEN)
# 设置成只读文件
win32api.SetFileAttributes(thefile, win32con.FILE_ATTRIBUTE_READONLY)
# 为了删除它先把它设成普通文件
win32api.SetFileAttributes(thefile, win32con.FILE_ATTRIBUTE_NORMAL)
# 最后删掉该文件
os.remove(thefile)
```

讨论

win32api.SetFileAttributes 的一个有趣的用法是用来删除文件。用 os.remove 在 Windows 中删除非普通文件会遭遇失败。为了删除文件，必须先通过 Win32 调用 SetFileAttributes

把该文件的属性设置为普通，如同本节代码最后一段所展示的那样。当然，这么做可能会遇到警告，因为一个文件没有被设置为普通文件通常是有很好的理由的。只有当你很明确并很有把握的时候，才应该真正的删掉一个文件。

更多资料

查看位于 http://ASPN.ActiveState.com/ASPN/Python/Reference/Products/ActivePython/PythonWin32Extensions/win32file.html 的 win32file 模块文档。

2.26 从 OpenOffice.org 文档中提取文本

感谢：Dirk Holtwick

任务

需要从 OpenOffice.org 文档的文本内容（无论有无 XML 标记）中抽取数据。

解决方案

OpenOffice.org 文档其实就是一个聚合了 XML 文件的 zip 文件，遵循一种良好的文档规范。如果只是为了访问其中的数据，我们甚至不用安装 OpenOffice.org：

```
import zipfile, re
rx_stripxml = re.compile("<[^>]*?>", re.DOTALL|re.MULTILINE)
def convert_OO(filename, want_text=True):
     """ 将一个 OpenOffice.org 文件转换成 XML 或文本 """
          zf = zipfile.ZipFile(filename, "r")
          data = zf.read("content.xml")
          zf.close( )
          if want_text:
               data = " ".join(rx_stripxml.sub(" ", data).split( ))
          return data
if __name__=="__main__":
     import sys
     if len(sys.argv)>1:
          for docname in sys.argv[1:]:
               print 'Text of', docname, ':'
               print convert_OO(docname)
               print 'XML of', docname, ':'
               print convert_OO(docname, want_text=False)
     else:
          print 'Call with paths to OO.o doc files to see Text and XML forms.'
```

讨论

OpenOffice.org 文档就是 zip 文件，并包含了一些其他内容，其中一般都会有 content.xml

文件。所以本节的任务其实就是从 zip 文件中抽取数据。本节的方法完全抛开了 XML 标记，只是用了一个简单的正则表达式，用空白符将内容切开，然后在片段间插入一个空格并最后合并起来，以节省空间。当然，我们也可以利用 XML 解析器，以更结构化的手段来获取信息，但是这里我们需要的只是一些文本内容，所以这种快速粗放的方法已经满足需要了。

特别指出一点，正则表达式 rx_stripxml 匹配的是 XML 标签（开始和结束）的起始符< 和结束符>。在函数 convert_OO 中，在 if want_text 语句之后，我们用这个正则表达式将所有的 XML 标签替换成空格，然后根据空白符进行切分（调用字符串方法 split，能够切割任何空白符序列），之后再合并（使用" ".join，用一个空格符作连接串）。这种切割再合并的方法，本质上是把任何空白符序列都变成一个空格符。更多的有关从 XML 文档中提取文本的内容可参看第 12.3 节。

更多资料

Library Reference 文档中的 zipfile 和 re 模块；OpenOffice.org 的主页 http://www.openoffice.org/；12.3。

2.27 从微软 Word 文档中抽取文本

感谢：Simon Brunning、Pavel Kosina

任务

你想从 Windows 平台下某个目录树中的各个微软 Word 文件中抽取文本，并保存为对应的文本文件。

解决方案

借助 PyWin32 扩展，通过 COM 机制，可以利用 Word 来完成转换：

```
import fnmatch, os, sys, win32com.client
wordapp = win32com.client.gencache.EnsureDispatch("Word.Application")
try:
     for path, dirs, files in os.walk(sys.argv[1]):
          for filename in files:
               if not fnmatch.fnmatch(filename, '*.doc'): continue
               doc = os.path.abspath(os.path.join(path, filename))
               print "processing %s" % doc
               wordapp.Documents.Open(doc)
               docastxt = doc[:-3] + 'txt'
               wordapp.ActiveDocument.SaveAs(docastxt,
                   FileFormat=win32com.client.constants.wdFormatText)
               wordapp.ActiveDocument.Close( )
```

```
finally:
     # 确保即使有异常发生 Word 仍能被正常关闭
     wordapp.Quit( )
```

讨论

关于 Windows 应用程序的一个有趣的地方是，可以通过 COM 以及 Python 提供的 PyWin32 扩展，编写一些简单的脚本对这些应用程序进行控制。这个扩展允许你用 Python 脚本来完成各种 Windows 下的任务。本节的脚本，从目录树下的所有的 Word 文档（即.doc 文件）中抽取文本，并存为对应的.txt 文本文件。通过使用 os.walk 函数，并利用 for 循环语句，我们无须递归即可遍历树中的所有子目录。通过 fnmatch.fnmatch 函数，可以检查文件名以确认它是否符合我们给出的通配符，这里的通配符是“.doc”。一旦我们确认了这是一个 Word 文档，我们就用此文件名和 os.path 来得到一个绝对路径，再用 Word 打开它，存为文本文件，然后关闭。

如果没有安装 Word，可能需要完全不同的方法来达成目标。一种可能是使用 OpenOffice.org，它也可以载入 Word 文档。另一种可能是使用可以读取 Word 文档的程序，比如 Antiword，其网址是 http://www.winfield.demon.nl/。但这里不准备探讨这两种方式。

更多资料

Mark Hammond、Andy Robinson 所著的 *Python Programming on Win32*（O’Reilly）一书中关于 PyWin32 的介绍；http://msdn.microsoft.com 介绍了微软 Word 的对象模型；*Library Reference* 和 *Python in a Nutshell* 中关于 fnmatch 和 os.path 以及 os.walk 的相关章节。

2.28 使用跨平台的文件锁

感谢：Jonathan Feinberg、John Nielsen

任务

希望某个能同时运行在 Windows 和类 UNIX 平台的程序具有锁住文件的能力，但 Python 标准库提供的锁定文件的方法却是平台相关的。

解决方案

如果 Python 标准库没有提供合适的跨平台解决方案，我们可以自己实现一个：

```
import os
# 需要 win32all 来工作在 Windows 下（NT、2K、XP、不包括 9x）
if os.name == 'nt':
     import win32con, win32file, pywintypes
```

```
        LOCK_EX = win32con.LOCKFILE_EXCLUSIVE_LOCK
        LOCK_SH = 0 # 默认
        LOCK_NB = win32con.LOCKFILE_FAIL_IMMEDIATELY
        __overlapped = pywintypes.OVERLAPPED( )
        def lock(file, flags):
            hfile = win32file._get_osfhandle(file.fileno( ))
            win32file.LockFileEx(hfile, flags, 0, 0xffff0000, __overlapped)
        def unlock(file):
            hfile = win32file._get_osfhandle(file.fileno( ))
            win32file.UnlockFileEx(hfile, 0, 0xffff0000, __overlapped)
    elif os.name == 'posix':
        from fcntl import LOCK_EX, LOCK_SH, LOCK_NB
        def lock(file, flags):
            fcntl.flock(file.fileno( ), flags)
        def unlock(file):
            fcntl.flock(file.fileno( ), fcntl.LOCK_UN)
    else:
        raise RuntimeError("PortaLocker only defined for nt and posix platforms")
```

讨论

当很多程序或线程需要访问一个共享的文件时，应该确保所有的访问是同步的，这样就不会出现两个进程或线程同时修改文件内容的情况。失败的同步访问在某些情况下会完全破坏掉整个文件。

本节代码给出了两个函数，lock 和 unlock，分别用于请求和释放一个文件的锁。对 portallocker.py 模块的使用其实就是简单地调用 lock 函数，传递一个文件给它，再用一个参数来指定需要的锁的类型：

`Shared lock （默认）`

这种锁会拒绝所有进程的写入请求，包括最初设定锁的进程。但所有的进程都可以读取被锁定的文件。

`Exclusive lock`

拒绝其他所有进程的读取和写入的请求。

`Nonblocking lock`

当这个值被指定时，如果函数不能获取指定的锁会立刻返回。否则，函数会处于等待状态。使用 Python 的位操作符，或操作（|），可以将 LOCK_NB 和 LOCK_SH 或 LOCK_EX 进行或操作。

举个例子：

```
import portalocker
afile = open("somefile", "r+")
portalocker.lock(afile, portalocker.LOCK_EX)
```

在不同系统中 lock 和 unlock 的实现完全不同。类 UNIX 系统（包括 Linux 和 Mac OS X）中，本节代码的功能依赖于标准的 fcntl 模块。在 Windows 系统中（NT、2000、XP。Win95 和 Win98 不适用，因为它们并没有提供相应的底层支持），则使用 win32file 模块，该模块是为 Windows 量身定做的流行的 PyWin32 扩展的一个组成部分，PyWin32 的作者是 Mark Hammond。重点是不管内部实现如何不同，这两个函数（包括需要传递给 lock 函数的标志）在不同的平台下却表现得完全一致。这种基于不同包的实现但却表现出功能一致性的方法，有助于写出跨平台的应用程序，这也正是 Python 的力量所在。

当写跨平台的程序时，最好是将各种功能以平台无关的方式封装起来。比如本节的文件锁部分，对于 Perl 用户是很有帮助的，他们很习惯直接使用系统调用 lock。更普遍的是，虽然 if os.name==不属于应用层面的代码，但是这类平台测试代码却总是应该被安插到标准库或者与应用无关的模块中。

更多资料

Library Reference 中 fcntl 模块的文档；http://ASPN.ActiveState.com/ASPN/Python/Reference/Products/ActivePython/PythonWin32Extensions/win32file.html 中有关 win32file 模块的介绍；Jonathan Feinberg 的主页（http://MrFeinberg.com）。

2.29 带版本号的文件名

感谢：Robin Parmar、Martin Miller

任务

如果你想在改写某文件之前对其做个备份，可以在老文件的名字后面根据惯例加上三个数字的版本号。

解决方案

我们需要编写一个函数来完成备份工作：

```
def VersionFile(file_spec, vtype='copy'):
     import os, shutil
     if os.path.isfile(file_spec):
          # 检查'vtype'参数
          if vtype not in ('copy', 'rename'):
               raise ValueError, 'Unknown vtype %r' % (vtype,)
          # 确定根文件名，所以扩展名不会太长
          n, e = os.path.splitext(file_spec)
          # 是不是一个以点为前导的三个数字？
          if len(e) == 4 and e[1:].isdigit( ):
```

```
                num = 1 + int(e[1:])
                root = n
            else:
                num = 0
                root = file_spec
            # 寻找下一个可用的文件版本
            for i in xrange(num, 1000):
                new_file = '%s.%03d' % (root, i)
                if not os.path.exists(new_file):
                    if vtype == 'copy':
                        shutil.copy(file_spec, new_file)
                    else:
                        os.rename(file_spec, new_file)
                    return True
            raise RuntimeError,"Can't%s%r,all names taken"%(vtype,file_spec)
        return False
if __name__ == '__main__':
        import os
        # 创建一个 test.txt 文件
        tfn = 'test.txt'
        open(tfn, 'w').close( )
        # 对它取 3 次版本
        print VersionFile(tfn)
        # 输出: True
        print VersionFile(tfn)
        # 输出: True
        print VersionFile(tfn)
        # 输出: True
        # 删除我们刚刚生成的 test.txt*文件
        for x in ('', '.000', '.001', '.002'):
            os.unlink(tfn + x)
        # 展示当文件不存在时取版本操作的结果
        print VersionFile(tfn)
        # 输出: False
        print VersionFile(tfn)
        # 输出: False
```

讨论

VersionFile 函数是为了确保在打开文件进行写入或更新等修改前，对已存在的目标文件完成了备份（或者重命名，由可选的第二个参数来决定）。在处理文件之前进行备份是很明智的举措（这也是一些人仍然怀念旧的 VMS 操作系统的原因，这种备份是自动进行的）。实际的复制和重命名是分别由 shutil.copy 和 os.rename 完成的，所以唯一的问题是，怎么确定文件的名字。

一个流行的决定备份的名字的方法是使之版本化（比如，给文件名增加一个逐渐增大

的数字)。本节确定文件名的方法是，首先从文件名中分解出名字根（因为有可能这已经是一个版本化的文件名了），然后在这个名字根之后添加进一步的扩展名，比如.000，.001，等等，直到以此种命名方式确定的文件名也无法对应任何一个存在的文件。注意，VersionFile 被限制为只能有 1 000 个版本，所以需要有个归档备份的计划。在进行版本化之前，首先要确保该文件存在——不能对不存在的东西进行备份。如果文件不存在，VersionFile 函数只是简单地返回 False（如果文件存在而且函数执行无误则返回 True），所以在调用之前也无须检查文件是否存在。

更多资料

Library Reference 和 *Python in a Nutshell* 中关于 os 和 shutil 模块的文档。

2.30 计算 CRC-64 循环冗余码校验

感谢：Gian Paolo Ciceri

任务

需要对某些数据进行循环冗余码校验（CRC）以确定数据的完整无误，而且必须遵循 ISO-3309 关于 CRC-64 校验的规定。

解决方案

Python 标准库并没有提供 CRC-64 的任何实现（但提供了 CRC-32 函数，即 zlib.crc32），所以我们需要自己来提供实现。幸而 Python 能够处理位操作（与、或、非、异或、移位等），就像 C 那样（实际上，它们的语法几乎完全相同），所以很容易根据 CRC-64 的参考实现写出如下的 Python 函数：

```
# 使用两个辅助表（为了速度我们用了一个函数），
# 之后删除该函数，因为我们再也用不着它了：
CRCTableh = [0] * 256
CRCTablel = [0] * 256
def _inittables(CRCTableh, CRCTablel, POLY64REVh, BIT_TOGGLE):
    for i in xrange(256):
        partl = i
        parth = 0L
        for j in xrange(8):
            rflag = partl & 1L
            partl >>= 1L
            if parth & 1:
                partl ^= BIT_TOGGLE
            parth >>= 1L
            if rflag:
                parth ^= POLY64REVh
```

```
            CRCTableh[i] = parth
            CRCTablel[i] = partl
# CRC64 的高 32 位的生成多项式（低 32 位被假设为 0）
# 以及_inittables 所用的 bit-toggle 掩码
POLY64REVh = 0xd8000000L
BIT_TOGGLE = 1L << 31L
# 运行函数来准备表
_inittables(CRCTableh, CRCTablel, POLY64REVh, BIT_TOGGLE)
# 删除我们不需要的名字，包括生成表的函数
del _inittables, POLY64REVh, BIT_TOGGLE
# 此模块公开了这两个函数：crc64 和 crc64digest
def crc64(bytes, (crch, crcl)=(0,0)):
     for byte in bytes:
          shr = (crch & 0xFF) << 24
          temp1h = crch >> 8L
          temp1l = (crcl >> 8L) | shr
          tableindex = (crcl ^ ord(byte)) & 0xFF
          crch = temp1h ^ CRCTableh[tableindex]
          crcl = temp1l ^ CRCTablel[tableindex]
     return crch, crcl
def crc64digest(aString):
     return "%08X%08X" % (crc64(bytes))
if __name__ == '__main__':
       # 当此模块作为主脚本运行时，一个小测试/展示
       assert crc64("IHATEMATH") == (3822890454, 2600578513)
       assert crc64digest("IHATEMATH") == "E3DCADD69B01ADD1"
       print 'crc64: dumb test successful'
```

讨论

循环冗余码校验（CRC）是一种流行的确保数据（例如，文件）未被损坏的方法。CRC 可以稳定地检查出一些随机偶发的损坏，但是对于恶意的针对性攻击并不像其他一些加密的校验和方法那么强劲。CRC 的计算比其他校验和方式都要快，因此在那些只需要检测偶发和随机损坏，而不用担心别人故意伪造数据进行欺骗的情况下，它比其他校验方式用得更多。

从数学的角度讲，CRC 是把需要校验和的数据的位当做多项式进行计算的。实际上，正如本节代码所示，经过正确的索引处理，计算可以一次完成并将结果存储在表中，数据中的每一个字节都对最终的结果产生影响。这样，在初始化之后（我们用一个辅助函数来初始化，这是因为在 Python 中使用局部变量进行计算要比使用全局变量计算快得多），CRC 计算的速度非常快。表的计算和相关的处理用到了很多位操作，不过，幸运的是，Python 对这类操作的处理效果和其他语言一样，比如 C，速度非常快。（实际上 Python 关于位操作的语法和 C 完全一样。）

这个标准的 CRC-64 校验和的算法在 ISO-3309 标准中有详细说明，本节的实现方法完全地遵照了该标准的描述。生成器多项式是 x64 + x4 + x3 + x + 1。（在“更多资料”小节中会提供更多的关于其计算方法的信息。）

用一对 Python 的 int 类型来存储 64 位的结果，分别代表其高 32 位和低 32 位。为了能够递进地计算 CRC——有时数据可能是逐步到达的，给 crc64 调用函数提供了一个可选的“初始值”，即(crch, crcl)构成的数对，这样可以在前一步计算结果的基础上继续计算。如果要一次计算出全部数据的 CRC，只需要提供完整的数据（字节的序列），如本节中一样，同时数对会被默认地设置为(0, 0)。

更多资料

W.H. Press，S.A. Teukolsky，W.T. Vetterling，以及 B.P. Flannery 所著的 Numerical Recipes in C, 2d ed. (Cambridge University Press), pp. 896ff。